Characteristics of a RTS

- Large and complex
- Concurrent control of separate system components
- **Facilities to interact with special purpose hardware**
- Guaranteed response times
- Extreme reliability
- Efficient implementation

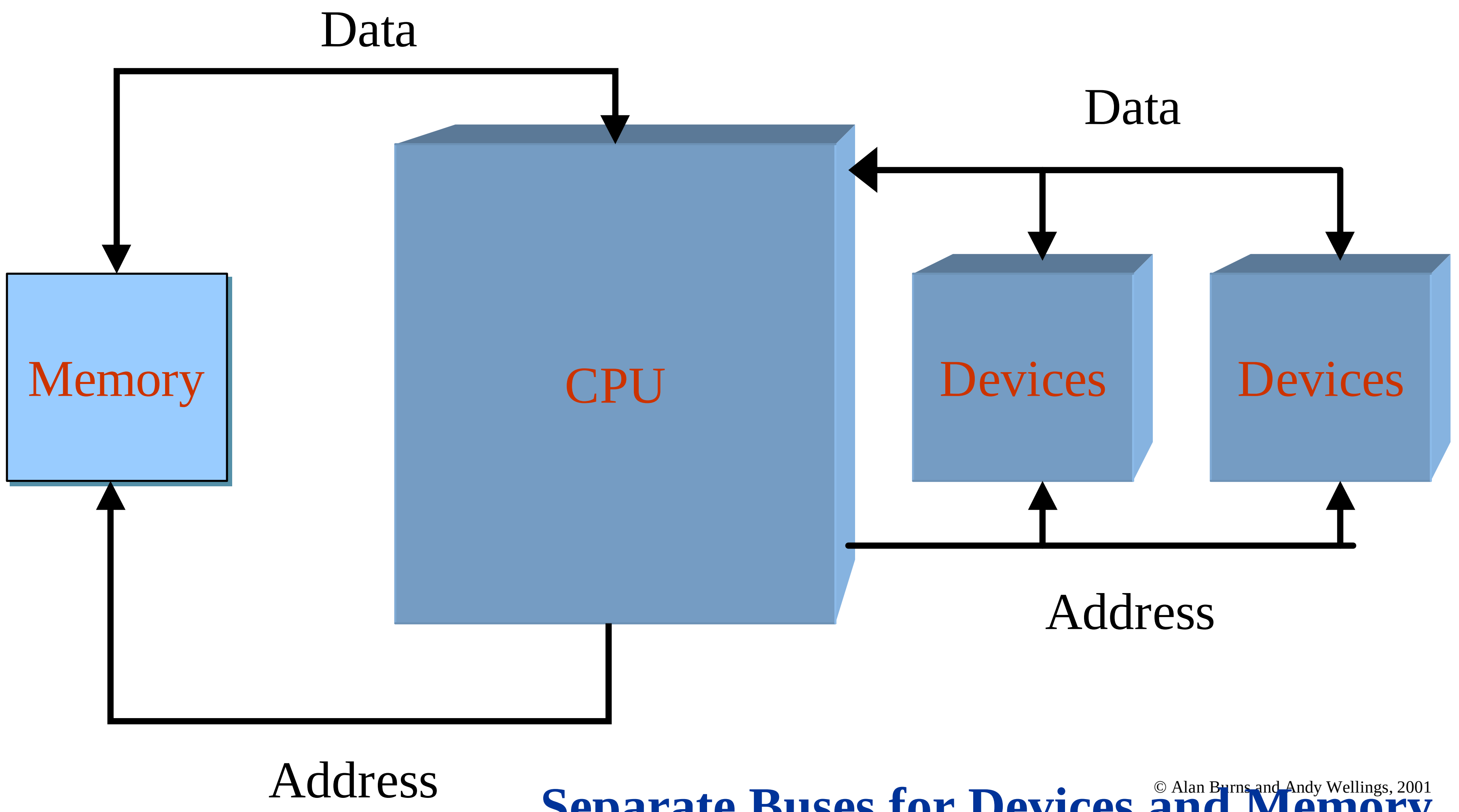
Low-level Programming

- Review hardware I/O mechanisms
- Look at language requirements and various models of device driving
- Consider the Modula-1, Ada, Real-Time Java and C models of device driving
- Memory Management

- Aim:
  - to show how the various models of processes and their communication and synchronisation mechanisms can be extended to allow devices to be modelled and controlled
  - to consider efficient real-time memory management

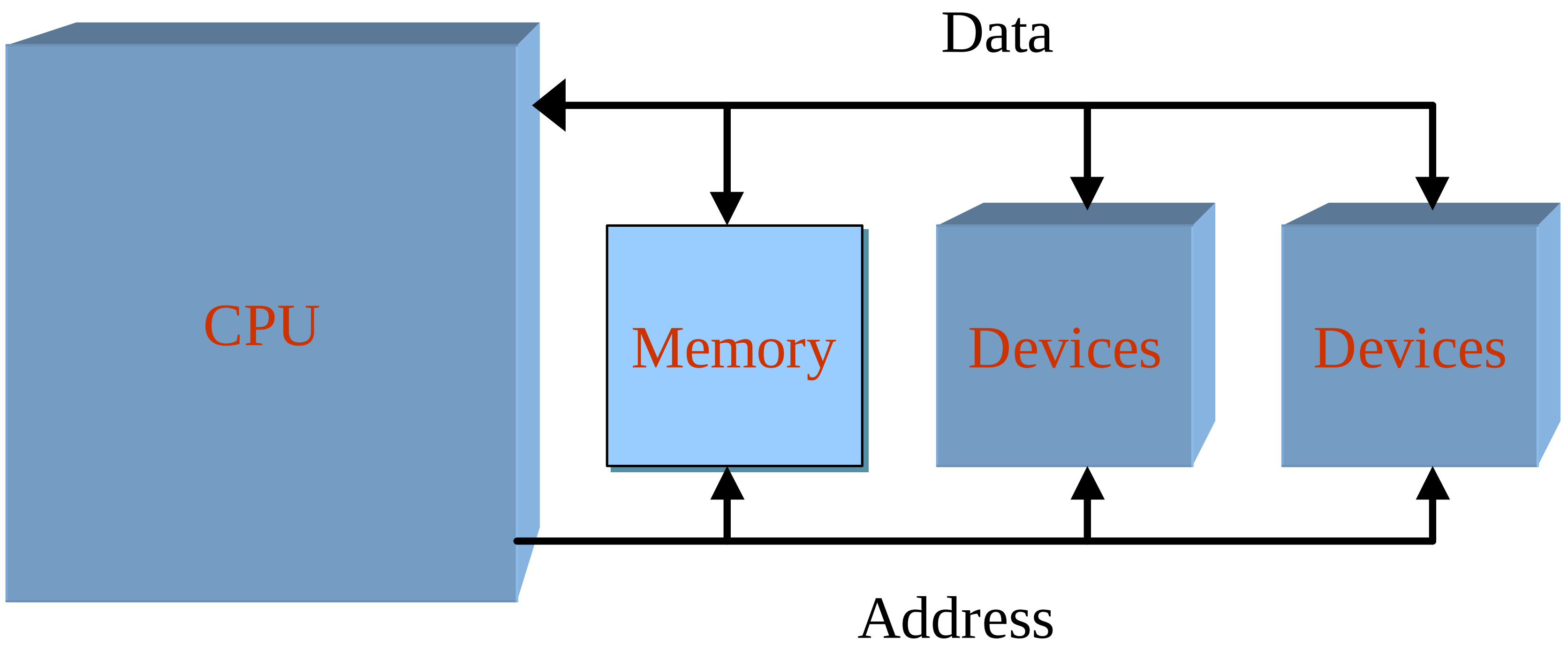
Hardware Input/Output Mechanisms

- Two general classes of computer architecture:



Separate Buses for Devices and Memory

Memory Mapped Architecture



Device Interface

- The interface to a device is normally through a set of registers
- Separate buses: two sets of assembly instructions — one for memory access the other for device register access
- The latter normally take the form of:

```
IN    AC, PORT
OUT   AC, PORT
```

- E.g., the Intel 486 and Pentium range
- Memory-mapped I/O: certain addresses access memory others the device registers; e.g., M68000 and PowerPC ranges
- The interface is used to control the device's operations and to control the data transfer
- Two control mechanisms: status driven and interrupt driven control

Status Driven

- A program performs explicit tests in order to determine the status of a given device
- Three kinds of hardware instructions:
  - test operations that enable the program to determine the status of the given device
  - control operations that direct the device to perform non-transfer device dependent actions such as positioning read heads
  - I/O operations that perform the actual transfer of data between the device and the CPU
- Nowadays most devices are interrupt driven. Interrupts can of course be turned off and polling of device status used instead.
- Interrupts are often no allowed in Safety Critical Systems

Interrupt Driven

- Interrupt-driven program-controlled
- Interrupt-driven program-initiated (DMA)
- Interrupt-driven channel-program controlled

DMA and channel programs can cause cycle stealing from the processor; this may make it difficult to estimate the worst-case execution time of a process

Elements Needed for Interrupt Driven Devices

1 Context switching mechanisms
  - Preserving the state (PC, registers, program status info - priority, memory protection etc.) of the processor immediately prior to the occurrence of the interrupt.
  - Placing the processor in the required state for processing the interrupt.
  - Restoring the suspended process state after the interrupt processing has been completed.
  - basic: just the PC is saved;
  - partial: PC and the PSW are saved;
  - complete: full context is saved.

- It may be necessary to supplement the actions of the hardware by explicit software support

Elements Needed

2 Interrupting device identification

- A vectored mechanism consists of a set of dedicated, contiguous memory locations (an interrupt vector) and a hardware mapping of device addresses onto the interrupt vector
- With a status mechanism, each interrupt has an associated status word which specifies the device causing the interrupt and the reason for the interrupt
- The polling device identification mechanism involves interrogating the status of each device

■ With some modern computer architectures, interrupt handling is directly associated with a high-level language primitive

■ With these systems, an interrupt is often viewed as a synchronisation message down an associated channel; the device is identified by the channel which becomes active

Elements Needed

3 Interrupt identification

- Once the device has been identified, the appropriate interrupt handling routine must determine why it generated the interrupt
- This can be supplied by either status information provided by the device or by having different interrupts from the same device occurring through different vectored locations or channels

Elements Needed

4 Interrupt control

- Once a device is switched on its interrupts must be enabled. Enabling/disabling of interrupts may be performed by:
  - Status mechanisms provide flags to enable/disable interrupts.
  - Mask interrupt control mechanisms associate device interrupts with particular locations in an interrupt mask word
  - Level-interrupt control mechanisms have devices associated with certain levels;the current level of the processor determines which devices may or may not interrupt

Elements Needed

5 Priority control

- Some devices have higher urgency than others and, therefore, a priority facility is often associated with interrupts
- This mechanism may be static or dynamic and is usually related to the device interrupt control facility and the priority levels of the processor

A Simple Example I/O System

- Loosely based on the Motorola 68000 series of computers; the registers are memory mapped
- Control & status registers contain all the information on a device's status, and allow the device's interrupts to be enabled / disabled.

bits	
15 - 12	: Errors
11	: Busy
10 - 8	: Unit select
7	: Done/ready
6	: Interrupt enable
5 - 3	: reserved
2 - 1	: Device function
0	: Device enable

A Simple Example I/O System

- Data buffer registers act as buffer registers for temporarily storing data to be transferred into or out of the machine via the device

15 - 8	: Unused
7 - 0	: Data

- A device may have more than one csr and dbr, the exact number being dependent on the nature of the device.
- On an interrupt, the processor stores the PC and the current program status word (PSW) on the system stack
- The new PC and PSW are loaded from an interrupt vector
- The first word contains the address of the interrupt service routine and the second contains the PSW including the priority at which the interrupt is to be handled

Language Requirements

- Modularity & encapsulation facilities
  - Device interfacing is machine dependent. It is important to separate the non-portable sections of code from the portable ones.
  - In Modula-1, devices are encapsulated in special modules.
  - In Ada, the package is used.
  - In Java, classes and packages
  - In C, it is a file
- An abstract model of device handling
  - A device can be viewed as a processor performing a fixed task. A computer system can be modelled as several parallel processes which need to communicate and synchronise; synchronisation is provided by the interrupt

Abstract Models

- All models require facilities for addressing and manipulating device registers
  - A device register may be represented as a program variable, an object, or even a communications channel
- A suitable representation of an interrupt:
  - procedure call
  - sporadic process invocation
  - asynchronous notification
  - shared-memory based condition synchronisation
  - message-based synchronisation
- All except procedure view the handler as executing in the scope of a process, and therefore require a full context switch

Abstract Models

- C/C++ use a procedural model with variables as device registers
- The Ada model is a hybrid between the procedural model and the shared memory model; protected procedure calls represent interrupts and variables are used for device register
- RTJ views an interrupt as an asynchronous event which is scheduled
- Modula-1 and RT Euclid used the shared memory model: Modula-1 maps condition variable to interrupts; RT Euclid uses semaphores
- Occam2 uses a message-based model

Modula-1

- A Pascal-like language with the addition of processes and modules
- A special type of module, called an interface module, has the properties of a monitor and is used to control access to shared resources
- Process interact via signals (condition variable) using the operators wait, send and awaited
- First language to attempt device driving in a high-level language
- A device module is a special type of interface module used to encapsulate the interaction with a device.
- It is only from within a device module that the facilities for handling interrupts etc. can be used

Modula-1: Modules

```
MODULE main;
  TYPE dimension = (xplane, yplane, zplane);

  PROCESS control (dim : dimension);
    VAR position : integer;     (* absolute position *)
    setting : integer;          (* relative movement *)
  BEGIN
    position := 0;              (* rest position *)
    LOOP
      new_setting (dim, setting);
      position := position + setting;
      move_arm (dim, position)
    END
  END control;
BEGIN
  control (xplane);
  control (yplane);
  control (zplane)
END main.
```

Modula-1: Hoare’s Monitors

```
INTERFACE MODULE resource_control;
  DEFINE allocate, deallocate;  (* export list *)
  VAR busy : BOOLEAN;
      free : SIGNAL;

  PROCEDURE allocate;
  BEGIN
    IF busy THEN WAIT(free) END;
    busy := TRUE;
  END;

  PROCEDURE deallocate;
  BEGIN
    busy := FALSE;
    SEND(free)
  END;
BEGIN (* initialisation of module *)
  busy := FALSE
END.
```

Addressing/Manipulating Device Registers

- Associating a variable with a register is expressed by an octal address following the name in a declaration.
- E.g. a data buffer register for the simple I/O architecture

```
VAR rdbr[177562B] CHAR;
```

  - 177562B is an octal address which is the location of the register in memory
- Only scalar data types can be mapped onto a device register; registers which have internal structures are considered to be of the predefined type bits whose definition is:

```
type bits = array 0:no_of_bits_in_word OF BOOLEAN;
```

- Variables of this type are packed into a single word

Addressing/Manipulating Device Registers

- A control and status register at octal address 177560 can therefore be defined by:

```
VAR rcsr[177560B] : BITS;
```

- The following will enable the device and turn interrupts off

```
rcsr[0] := TRUE;
rcsr[6] :=False;
```

- In general these facilities are not powerful enough to handle all types of register conveniently; consider setting

  10 - 8 : Unit select

- to the value 5

```
rcsr[10] := true;
rcsr[9]  := false;
rcsr[8]  := true;
```

Clumsy

Note

- On many machines more than one device register can be mapped to the same physical address; these registers are often read or write only
- Care must be taken when manipulating device registers; If the CSR was a pair of registers mapped to the same location, the code will not have the desired affect WHY?
- It is advisable to have other variables in a program which represent device registers; these can be manipulated in the normal way
- When the required register format has been constructed it may then be assigned to the actual device register
- Such variables are often called shadow device registers

Interrupt Handling in Modula-1

- Based around the concept of an ideal hardware device with the following properties
  - Each device operation is known to produce either no interrupt or at least one
  - After an interrupt has occurred the device status indicates whether or not another interrupt will occur
  - No interrupt arrives unexpectedly
  - Each device has a unique interrupt location
- The facilities provided:
  - Each device has an associated device module
  - Each device module has a hardware priority specified in its header following the module name
  - All code within the module executes at the specified hardware priority

Modula-1 Facilities

- Each interrupt to be handled within a device module requires a process called a device process
- When the device process is executing it has sole access to the module (i.e. it holds the monitor lock)
- A device process is not allowed to call any non-local procedures and cannot send signals to other device processes; this is to ensure that device processes will not be inadvertently blocked
- When a device process sends a signal, the receiving process is not resumed but the signalling process continues (this is to ensure that the device process is not blocked)
- Wait statements within device processes may only be of rank 1

Modula-1: Interrupts

- An interrupt is considered to be a form of signal, the device process, however, instead of issuing a wait request issues a DOIO request
- The address of the vector through which the device interrupts is specified in the header of the process
- Only device processes can contain DOIO statements
- DOIO and wait calls lower the processor priority and therefore release the monitor lock
- Only one instance of a device process may be activated

Clock Handler in Modula-1

```
DEVICE MODULE rtc[6]; (* hardware priority 6 *)
  DEFINE tick;
  VAR tick : SIGNAL;
  PROCESS clock[100B]; (* interrupt vector address *)
    VAR csr[177546B] : BITS; (*CSR *)
  BEGIN
    csr[6] := TRUE;  (* enable interrupts *)
    LOOP
      DOIO; (* wait for interrupts *)
      WHILE AWAITED(tick) DO
        SEND(tick);
      END
    END
  END;
BEGIN
    clock; (* create one instance of the clock process *)
END rtc;
```

Modula-1 and Interrupt-Driven Devices

- Device control — I/O registers are represented by variables
- Context switching — the interrupt causes an immediate context switch to the handling process, which waits using the DOIO
- Interrupt device identification — the address of the interrupt vector is given with the device process' header
- Interrupt identification — in general the device status register should be checked to identify the cause of the interrupt
- Interrupt control — the interrupt control is status driven and provided by a flag in the device register
- Priority control — the priority of the device is given in the device module header; all code in the module runs at this priority

An Example Terminal Driver

```
DEVICE MODULE Keyboard[4];
  DEFINE readch;
  CONST size=64;              (* buffer size *)
  VAR KBS[177560B]: BITS; (* keyboard status *)
  KBB[177562B]: CHAR; (* keyboard buffer *)
  in, out, n : INTEGER;
  nfull, nempty : SIGNAL;
  buf : ARRAY 1:n OF CHAR;
```

```
PROCEDURE readch(VAR ch : CHAR);
BEGIN
  IF n = 0 THEN WAIT(nempty) END;
  ch := buf[out];
  out := (out MOD size)+1;
  DEC(n);
  SEND(nfull)
END readch;
```

The buffers must be included in the device module because device processes cannot call non-local procedures

```
  PROCESS keyboarddriver[60B];
  BEGIN
    LOOP
      IF n1 = n THEN WAIT(nfull) END;
      KBS[6] := TRUE;
      DOIO;
      KBS[6] := FALSE;
      buf[in] := KBB;
      in := (in MOD size)+1;
      INC(n);
      SEND(nempty)
    END
  END keyboarddriver;
BEGIN
      in :=1; out :=1; n :=0;
      keyboarddriver;
END terminal.
```

Example in book, has display handler in same module

Timing Facilites

```
DEVICE MODULE hardwareclock[6];
  DEFINE tick;
  VAR tick : SIGNAL;

  PROCESS handler[100B];
    VAR count : INTEGER;  statusreg[177546B] : BITS;
  BEGIN
    count := 0; statusreg[6] := TRUE;
    LOOP
      DOIO; count := (count+1) MOD 50;
      IF count = 0 THEN
        WHILE AWAITED(tick) DO SEND(tick) END
      END
    END
  END handler;
BEGIN
    driver
END hardwareclock;
```

```
INTERFACE MODULE SystemClock;
  DEFINE GetTime, SetTime;
  USE time, initialise, add, tick;
  VAR TimeOfDay, onesec : time;

  PROCEDURE SetTime(t: time);
  BEGIN TimeOfDay := t; END SetTime;

  PROCEDURE GetTime(VAR t: time);
  BEGIN t := TimeOfDay END GetTime;

  PROCESS clock;
  BEGIN
    LOOP
      WAIT(tick);
      addtime(TimeOfDay, onesec)
    END
  END clock;
BEGIN
    inittime(TimeOfDay, 0, 0, 0);
    inittime(onesec, 0, 0, 1);
    clock;
END SystemClock;
```

The clock process is logically redundant. The device process could increment TimeOfDay directly thereby saving a context switch. However, Modula-1 forbids a device process to call a non-local procedure.

Problems with the Modula-1 Approach

- It does not allow a device process to call a non-local procedure; therefore you have to include extra functions into a device module (e.g. bounded buffer in terminal driver), or introduce extra processes to wait for a signal sent by a device process (e.g. clock)
- It only allows a single instance of a device process; this makes the sharing of code between similar devices difficult; the problem is compounded by not being able to call non-local procedures
- Modula-1 was design for memory mapped machines and consequently it is difficult to use its facilities machines with special instructions

```
VAR x AT PORT 46B : INTEGER;
```

- It is not possible to define variables that are read/write-only in Modula-1

Interrupt Handling and Device Driving in Ada

- A device driver is a subsystem which has responsibility for controlling access to some external device; it must manipulate device registers and respond to interrupts

- The device can be modeled as a hardware task

- There are 3 ways in which tasks can communicate and synchronise:
  1. through the rendezvous
  2. using protected units
  3. via shared variables

Interrupt Handling II

- Ada assumes that shared memory device registers can be specified using representation specifications

- In Ada 83 an interrupt was treated as a hardware entry call, Ada 95 prefers it to be viewed as a hardware protected procedure call

Ada: Addressing and Manipulating Device Registers

- Ada has a comprehensive set of facilities for specifying the implementation of data types
- These are collectively known as Representation clauses
- A representation clause can be
  - attribute definition clause: size, alignment, storage space for tasks, address
  - enumeration representation clause: internal values for literals
  - record representation clause: offsets and lengths of components
  - address (at) clause: Ada 83 - obsolete

Example of Representation Clauses

```
type Error_T is (Read_Error, Write_Error,
                 Power_Fail, Other);
type Function_T is (Read, Write, Seek);
type Unit_t is new Integer range 0 .. 7;

type Csr_T is record
  Errors        : Error_T;
  Busy          : Boolean;
  Unit          : Unit_T;
  Done         : Boolean;
  Ienable      : Boolean;
  Dfun         : Function_T;
  Denable    : Boolean;
end record;
```

Device registers represented as a user-defined record structure

Enumeration Clause

- specifies the internal codes for the literals of the enumeration type

01	-	Read
10	-	Write
11	-	Seek

```
type Function_T is (Read, Write, Seek);
for Function_T use (Read=>1,Write=>2,Seek=>3);
```

Record Representation Clause

- Specifies the storage representation of records; that is, the order, position and size of its components
- The bits in the record are numbered from 0; the range in the component clause specifies the number of bits to be allocated
- There are also size, alignment and bit ordering attributes

```
Word : constant :=2; --no. of bytes in a word
Bits_In_Word : constant := 16;
for Csr_T use
  record
    Denable  at 0*Word range 0..0;
    Dfun     at 0*Word range 1..2;
    Ienable  at 0*Word range 6..6;
    Done     at 0*Word range 7..7;
    Unit     at 0*Word range 8 .. 10;
    Busy     at 0*Word range 11 .. 11;
    Errors   at 0*Word range 12 .. 15;
  end record;
for Csr_T’Size use Bits_In_Word;
for Csr_T’Alignment use Word;
for Csr_T’Bit_Order use Low_Order_First;
```

Register Definition and Use

```
Tcsr : Csr_T;
for Tcsr'Address use
System.Storage_Elements.To_Address( 8#177566#);
Tmp :Csr_T;

-- The hardware register can be manipulated:
Tmp := (Denable => True, Dfun => Read,
     Ienable => True, Done => False,
     Unit => 4, Errors => None );
Tcsr := Tmp; -- to ensure all bits are set at once

-- To test for errors
if Tcsr.Error = Read_Error then
  raise Disk_Error;
end if;
```

System

```
package System is
  pragma Preelaborate(System);

  -- storage-related declarations
  type Address is implementation-defined;
  Null_Address : constant Address;
  Storage_Unit : constant := implementation-defined;
  Word_Size    : constant :=
       implementation-defined * Storage_Unit;
  Memory_Size  : constant := implementation-defined;


  -- address comparison
  function "<" (Left, Right : Address) return Boolean;
  -- similarly for "<=”, ">”, "="
  pragma Convention(Intrinsic, "<");
  -- similarly for all subprograms in this package
```

System II

```
  -- other system-dependent declarations
  type Bit_Order is (High_Order_First, Low_Order_First);
  Default_Bit_Order : constant Bit_Order;

  -- priority-related declarations
  subtype Any_Priority is Integer range implementation-defined;
  subtype Priority is Any_Priority range
                    Any_Priority'First .. implementation-defined;
  subtype Interrupt_Priority is Any_Priority range
                    Priority'Last+1 .. Any_Priority'Last;

  Default_Priority : constant Priority :=
                   (Priority'First + Priority'Last)/2;

private
  -- not specified by the language
end System;
```

Storage Elements

```
package System.Storage_Elements is
  pragma Preelaborate(System.Storage_Elements);

  type Storage_Offset is range implementation-defined;

  subtype Storage_Count is Storage_Offset range
                               0..Storage_Offset'Last;

  type Storage_Element is mod implementation-defined;
  for Storage_Element'Size use Storage_Unit;

  type Storage_Array is array
      (Storage_Offset range <>) of aliased Storage_Element;
  for Storage_Array'Component_Size use Storage_Unit;

  -- Address Arithmetic, including:
  function "+"(Left : Address; Right : Storage_Offset)
                 return Address;
  function "+"(Left : Storage_Offset; Right : Address)
                 return Address;
```

Storage Elements II

```
  function "mod"(Left : Address; Right : Storage_Offset)
                return Storage_Offset;

  -- Conversion to/from integers:
  type Integer_Address is implementation-defined;
  function To_Address(Value : Integer_Address) return Address;
  function To_Integer(Value : Address) return Integer_Address;

  pragma Convention(Intrinsic, "+");
  -- ...and so on for all language-defined subprograms
  -- declared in this package.
end System.Storage_Elements;
```

Interrupt Model

- An interrupt represents a class of events that are detected by the hardware or systems software
- The **occurrence** of an interrupt consists of its **generation** and its **delivery**
- The generation of an interrupt is the event in the underlying hardware or system which makes the interrupt available to the program
- Delivery is the action which invokes a part of the program (the interrupt handler) in response to the interrupt occurrence; in between its generation and its delivery, the interrupt is pending
- The latency is the time spent in the pending state
- The handler is invoked once per delivery

Interrupt Model II

- When an interrupt is being handled, further interrupts from the same source are blocked
- It is device dependent if a blocked interrupt remains pending or is lost
- Certain interrupts are reserved (e.g. clock interrupt used to implement the delay statement)
- Each non-reserved interrupt has a default handler assigned by the RTS
- Each interrupt has an implementation-defined unique identifier supported by the system (e.g. address of the interrupt vector)

Interrupts and Protected Procedures

- Identifying an interrupt handler is done by using one of two pragmas

```
pragma Interrupt_Handler(Handler_Name);
```

This can appear in the specification of a library level protected unit and allows the dynamic association of the named parameterless procedure as an interrupt handler for one or more interrupts. Objects created from a type must be library-level.

```
pragma Attach_Handler(Handler_Name, Expression);
```

This can appear in the specification or body of a library-level protected unit and allows the association of the named handler with the interrupt identified by the expression; the handler becomes attached when the protected object is created. Can raise `Program_Error`.

Attachment of Interrupt Handlers

```
package Ada.Interrupts is
  type Interrupt_Id is implementation_defined;
       --discrete type
  type Parameterless_Handler is
       access protected procedure;

  function Is_Reserved(Interrupt : Interrupt_Id)
                     return Boolean;

  function Is_Attached (Interrupt : Interrupt_Id)
                      return Boolean;
    -- Raises Program_Error if interrupt is reserved

  function Current_Handler (Interrupt : Interrupt_Id)
                       return Parameterless_Handler;
    -- Raises Program_Error if interrupt is reserved
```

Attachment of Interrupt Handlers II

```
procedure Attach_Handler
     (New_Handler : Parameterless_Handler;
      Interrupt : Interrupt_Id);
```

Raises `Program_Error` if the protected object associated with `New_Handler` has not been identified with a pragma `Interrupt_Handler`, or interrupt is reserved, or if current handler was statically attached using the `Attach_Handler` pragma

Attachment of Interrupt Handlers III

```
  procedure Exchange_Handler
    (Old_Handler : out Parameterless_Handler;
     New_Handler : Parameterless_Handler;
     Interrupt : Interrupt_Id);
     -- Raises Program_Error as above

  procedure Detach_Handler(Interrupt : Interrupt_Id);
    -- Raises Program_Error if interrupt is reserved

  function Reference(Interrupt : Interrupt_Id)
      return Address;
      -- returns an address for use
      -- in a task entry address clause
end Ada.Interrupts;
```

Interrupt Names

```
package Ada.Interrupts.Names is
  implementation_defined : constant Interrupt_Id :=
                                    implementation_defined;
   ...
  implementation_defined : constant Interrupt_Id :=
                                    implementation_defined;
private
  -- not specified by the language
end Ada.Interrupts.Names;
```

A Simple Device Driver ¾ An ADC

- a 16 bit result register at 8#150000#
- a 16 bit control register at 8#150002#

Bit	Name	Meaning
0	A/D Start	Set to 1 to start a conversion
6	Interrupt/Enable/ Disable	Set to 1 to enable the device
7	Done	Set to 1 when conversion complete
8-13	Channel	Required input channel out of 64
15	Error	Set if device malfunctions

Driver: Package Specification

```
package Adc_Device_Driver is
  Max_Measure :constant := (2**16)-1;
  type Channel is range 0 .. 63;
  subtype Measurement is Integer range 0 .. Max_Measure;
  procedure Read (Ch: Channel; M : out Measurement);
    -- potentially blocking
  Conversion_Error : exception;
private
  for Channel'Size use 6; -- only six bits
end Adc_Device_Driver;
```

Driver: Package Body

```
with Ada.Interrupts.Names, System;
with System.Storage_Elements;
use Ada.Interrupts.Names, System;
use System.Storage_Elements;
package body Adc_Device_Driver is

  Bits_In_Word : constant := 16;
  Word : constant 2; -- 2 bytes in a word
  type Flag is (Down, Set);
  for Flag use (Down => 0; Set => 1);
  type Control_Register is
    record
      Ad_Start     : Flag;
      ie           : Flag;
      Done         : Flag;
      Ch           : Chan;
      Error        : Flag;
    end record;
```

```
for Control_Register use
   record
     Ad_Start     : at 0 range 0 .. 0;
     IE           : at 0 range 6 .. 6;
     Done         : at 0 range 7 .. 7;
     Ch           : at 0 range 8 .. 13;
     Error        : at 0 range 15 .. 15;
   end record;

 for Control_Register'Size use Bits_In_Word;
 for Control_Register'Alignment use Word;
 for Control_Register’Bit_Order use Low_Order_First;

 type Data_Register is range 0 .. Max_Measure;
 for Data_Register'Size use Bits_In_Word;
```

```
Control_Reg_Addr : constant Address :=
        Storage_Elements.To_Address(8#150002#);
Data_Reg_Addr : constant Address :=
        Storage_Elements.To_Address(8#150000#);
Adc_Priority : constant Interrupt_Priority := 63;


Control_Reg : aliased Control_Register;
for Control_Reg'Address use Control_Reg_Addr;
Data_Reg : aliased Data_Register;
for Data_Reg'Address use Data_Reg_Addr;
```

```
protected type Interrupt_Interface(
           Int_Id : Interrupt_Id;
           Cr : access Control_Register;
           Dr : access Data_Register) is
  entry Read(Chan : Channel; M : out Measurement);
private
  entry Done(Chan : Channel; M : out Measurement);
  procedure Handler;
  pragma Attach_Handler(Handler, Int_Id);
  pragma Interrupt_Priority(Adc_Priority);
  Interrupt_Occurred : Boolean := False;
  Next_Request : Boolean := True;
end Interrupt_Interface;
```

```
Adc_Interface:Interrupt_Interface(Names.Adc_Interrupt,
                 Control_Reg'Access, Data_Reg'Access);
```

```
protected body Interrupt_Interface is
  entry Read(Chan : Channel; M : out Measurement)
    when Next_Request is
   Shadow : Control_Register;
  begin
    Shadow := (Ad_Start => Set, IE => Set,
       Done => Down, Ch => Chan, Error => Down);
    Cr.all := Shadow;
    Interrupt_Occurred := False;
    Next_Request := False;
    requeue Done;
  end Read;

  procedure Handler is
  begin
    Interrupt_Occurred := True;
  end Handler;
```

```
entry Done(Chan : Channel; M : out Measurement)
       when Interrupt_Occurred is
   begin
     Next_Request := True;
     if Cr.Done = Set and Cr.Error = Down then
       M := Measurement(Dr.all);
     else
       raise Conversion_Error;
     end if;
   end Done;
 end Interrupt_Interface;
 procedure Read(Ch : Channel; M : out Measurement) is
  begin
    for I in 1 .. 3 loop
      begin
        Adc_Interface.Read(Ch, M);
        return;
      exception
        when Conversion_Error => null;
      end;
    end loop;
    raise Conversion_Error;
  end Read;
end Adc_Device_Driver;
```

Dynamic Attachment of Handlers

- To change dynamically the interrupt handler, requires that the definition be changed:
- Here the pragma now indicates the intention for Handler to be used as an interrupt handler

```
protected type Interrupt_Interface (
             Cr : access Control_Register;
             Dr : access Data_Register) is
  entry Read(Ch : Channel; M : out Measurement);
  procedure Handler;
  pragma Interrupt_Handler(Handler);
private
  entry Done(Ch : Channel; M : out Measurement);
  pragma Interrupt_Priority(Interrupt_Priority);
  -- register declaration etc
end Interrupt_Interface;
```

Dynamic Attachment of Handlers II

```
New_Adc_Interface : New_Interrupt_Interface(
             Control_Reg'Access, Data_Reg'Access);

Old : Parameterless_Handler := null;
...
-- attach new handler
if Is_Attached(Names.Adc) then
  Exchange_Handler(Old, New_Adc_Interface.Handler'access,
                   Names.Adc);
else
  Attach_Handler(New_Adc_Interface.Handler'access,
                   Names.Adc);
end if;
...
if Old = null then
  Detach(Names.Adc);
else
  Attach_Handler(Old, Names.Adc);
end if;
```

Real-Time Java

- RTJ allows access to memory mapped device registers via the concept of raw memory
- An implementation is allowed to support a range of memory types, e.g. DMA, shared memory, IO_Page

```
public class RawMemoryAccess {
  protected RawMemoryAccess(long base, long size);
  protected RawMemoryAccess(RawMemoryAccess memory,
                            long base, long size);

  public static RawMemoryAccess create(
                java.lang.Object type, long size)
      throws SecurityException, OffsetOutOfBoundsException,
             SizeOutOfBoundsException,
             UnsupportedPhysicalMemoryException;

  public static RawMemoryAccess create(
         java.lang.Object type, long base, long size)
      throws SecurityException, OffsetOutOfBoundsException,
               SizeOutOfBoundsException,
               UnsupportedPhysicalMemoryException;

  public byte getByte(long offset)
     throws SizeOutOfBoundsException, OffsetOutOfBoundsException;
  // similarly for integers, long integers, etc

  public void setByte(long offset, byte value)
     throwsSizeOutOfBoundsException, OffsetOutOfBoundsException;
  // similarly for integers, long integers etc
}
```

Control and Status Register

```
public class ControlAndStatusRegister
{

  RawMemoryAccess rawMemory;

  public ControlAndStatusRegister(long base, long size)
  {
    rawMemory = RawMemoryAccess.create(IO_Page, base, size);
  }

  public void setControlWord(short value)
  {
    rawMemory.setShort(0, value);
  }

};
```

Using the CSR

```
{
  byte shadow, channel;
  final byte start = 01;
  final byte enable = 040;
  final long csrAddress = 015002;
  final long csrSize = 2;
  ControlAndStatusRegister csr = new
           ControlAndStatusRegister(csrAddress, csrSize);

  channel = 6;
  shadow = (channel << 8) | start | enable;
  csr.setControlWord(shadow);
}
```

Interrupt Handling

- RTJ views an interrupt as an asynchronous event
- The interrupt is equivalent to a call of the `fire` method
- The association between the interrupt and the event is achieved via the `bindTo` method in the `AsyncEven`t class
- The parameter is of string type, and this is used in an implementation-dependent manner — one approach might be to pass the address of the interrupt vector
- When the interrupt occurs, the appropriate handler's fire method is called
- Now, it is possible to associate the handler with a schedulable object and give it an appropriate priority and release parameters

Interrupt Handling

```
AsyncEvent Interrupt = new AsyncEvent();
AsyncEventHandler InterruptHandler = new
      BoundAsyncEventHandler(
           priParams, releaseParams, null, null, null);

Interrupt.addHandler(InterruptHandler);
Interrupt.bindTo(”0177760");
```

Device Driving in C

- Device registers are addressed by pointer variables which can be assigned to the memory location of the register
- They are manipulated by low-level bitwise logical operators
- For example, the following procedure assigns n bits starting at position p in register pointed at by reg to x

```
unsigned int setbits(unsigned int *reg, unsigned int n,
             unsigned int p, unsigned int x)
{
  unsigned int data, mask;

  data = (x & (~(~0 << n))) << (p); /* data to be masked in */
  mask = ~(~0 << n); /* mask */
  *reg &= ~(mask << (p)); /* clear current bits */
  *reg |= data; /* or in data */
}
```

Interrupt Handling in C

- With the simple I/O architecture, interrupts handlers are assigned by placing the address of a parameterless procedure in the appropriate interrupt vector location
- Once the procedure is executed, any communication and synchronization with the rest of the program must be programmed directly
- Although POSIX provides alternative mechanisms which, in theory, could be used to provide an alternative model of interrupt handling (for example, associating an interrupt with a condition variable), there is currently no standard mechanism for attaching user-defined handlers to interrupts

Memory Management

- Embedded RTS often have a limited amount of memory
- This is due to: cost, or size, power or weight constraints)
- It is necessary to control how this memory is allocated so that it can be used effectively
- Where there is more than one type of memory (with different access characteristics), it is necessary to instruct the compiler to place certain data types at certain locations (e.g. Ada’s Representation Specs)
- The more general issue is of storage management of the
  - heap
  - stack

Heap Management

- For use with allocators (the new operator)
- Key problems:
  - how much space is required (requires application knowledge)
  - when can allocated space be released
- Returning allocated space
  - require the programmer to do it (`malloc`, `free`, `sizeof` in C) ; error prone
  - require the run-time to monitor memory and determine when it logically can no longer be accessed (the scope rules of Ada and Real-Time Java allow this)
  - require the run-time to monitor memory and release it when it it is no longer being used (garbage collection in Java)

Real-Time Perspective

- These approaches have an increasing impact on the ability to analyse the timing properties of the program
- In particular, garbage collection may be performed either when the heap is empty or by an asynchronous activity (incremental garbage collection)
- In either case, running the garbage collector may have a significant impact on the response time of a time-critical task
- Although there has been much work on real-time garbage collection and progress continues to be made, there is still a reluctance to rely on these techniques in time-critical systems

Heap Management in Ada

- The heap is represented by one or more storage pools
- Each object (access type) has an associated storage pool
- The allocator takes its memory from the target pool
- The `Ada.Unchecked_Deallocation` facility returns data to the pool
- An implementation may support
  - a single global pool (reclaimed when the program terminates)
  - pools defined at different accessibility levels (reclaimed when associated scope is exited)
- Note, all objects accessed directly (not via a pointer) are placed on the stack, not the heap.
- To give more user control over storage management, Ada defines a package called `System.Storage_Pools`

```
with Ada.Finalization; with System.Storage_Elements;
package System.Storage_Pools is
  pragma Preelaborate(System.Storage_Pools);
  type Root_Storage_Pool is abstract new
       Ada.Finalization.Limited_Controlled with private;
  procedure Allocate(Pool : in out Root_Storage_Pool;
            Storage_Address : out Address;
            Size_In_Storage_Elements : in System.
                 Storage_Elements.Storage_Count;
            Alignment : in System.Storage_Elements.Storage_Count)
            is abstract;
  procedure Deallocate(Pool : in out Root_Storage_Pool;
            Storage_Address : in Address;
            Size_In_Storage_Elements : in System.
                 Storage_Elements.Storage_Count;
            Alignment : in System.Storage_Elements.Storage_Count)
            is abstract;
  function Storage_Size(Pool : Root_Storage_Pool) return
           System.Storage_Elements.Storage_Count is abstract;
private ...
end System.Storage_Pools;
```

Storage Pools

- Programmers can implement their own storage pools by extending the `Root_Storage_Pool` type and providing concrete implementations for the subprogram bodies
- To associate an access type with a storage pool, the pool is declared and then the Storage_Pool attribute is used:

```
My_Pool : Some_Storage_Pool_Type;

type A is access Some_Object;
for A'Storage_Pool use My_Pool;
```

- Calls to **new** using A will automatically call `Allocate`; calls to `Ada.Unchecked_Deallocation` will call `Deallocate`; both referring to `My_Pool`
- `Deallocate` is called when A goes out of scope
- Note, Ada does not require an implementation to support garbage collection

Heap Management in Real-Time Java

```
public abstract class MemoryArea {
  protected MemoryArea(long sizeInBytes);

  public void enter(java.lang.Runnable logic);
  // associate this memory area to the current thread
  // for the duration of the logic.run method

  public static MemoryArea getMemoryArea(java.lang.Object object);
  // get the memory area associated with the object

  public long memoryConsumed();
  // number of bytes consumed in this memory area
  public long memoryRemaining();
  // number of bytes remaining

  . . .

  public synchronized java.lang.Object newInstance(
         java.lang.Class type)throws IllegalAccessException,
            InstantiationException, OutOfMemoryError;
  // allocate an object

  public long size(); // the size of the memory area
}
```

Immortal Memory

- Immortal memory is shared among all threads in an application
- Objects created in immortal memory are never subject to garbage collection and are freed only when the program terminates

```
public final class ImmortalMemory extends MemoryArea
{
  public static ImmortalMemory instance();
}
```

- There is also a class called `ImmortalPhysicalMemory` which has the same characteristics as immortal memory but allows objects to be allocated from within a range of physical addresses

Scoped Memory

- A memory area where objects which have a well-defined lifetime
- May be entered explicitly (by the use of the `enter` method) or implicitly by attaching it to a `RealtimeThread` at thread creation time
- Associated with each scoped memory is a reference count which incremented for every call to `enter` and at every associated thread creation
- It is decremented when the `enter` method returns and at every associated thread exit
- When the reference count reaches 0, all objects resident in the scoped memory have their finalization method executed and the memory is reclaimed
- Scoped memory can be nested by nested calls to `enter`

Scoped Memory

```
public abstract class ScopedMemory extends MemoryArea
{
  public ScopedMemory(long size);

  public void enter(java.lang.Runnable logic);

  public int getMaximumSize();

  public MemoryArea getOuterScope();

  public java.lang.Object getPortal();

  public void setPortal(java.lang.Object object);
}
```

Scoped Memory

- The `ScopedMemory` class which has several subclasses
  - `VTMemory`: allocations may take variable amounts of time
  - `LTMemory`: allocations occur in linear time (related to the size of the object)
  - `ScopedPhysicalMemory`: allowing objects to be allocated at physical memory locations
- To avoid the possibility of dangling pointers, a set of access restrictions are placed on the use of the various memory areas
  - Heap objects -- can reference other heap objects and objects in immortal memory only (i.e. it cannot access scoped memory)
  - Immortal objects -- can reference heap objects and immortal memory objects only;
  - Scoped objects -- can reference heaped objects, immortal objects and objects in the same scope or an outer scope only

Example

```
import javax.realtime.*;
public class ThreadCode implements Runnable
{
  private void computation()
  {
    final int min = 1*1024;
    final int max = 1*1024;
    final LTMemory myMem = new LTMemory(min, max);

    myMem.enter(new Runnable()
    {
      public void run()
      {
        // code here which requires access
        // to temporary memory
      }
    } );
  }
```

Example

```
public void run()
  {
    ...
    computation();
    ...
  }
}
```

- The thread can now be created; note, no parameters other than the memory area and the `Runnable` are given

```
ThreadCode code = new ThreadCode();

RealtimeThread myThread = new RealtimeThread(
        null, null, null, ImmortalMemory.instance(),
        null, code);
```

Stack Management

- Embedded programmers also have to be concerned with stack size
- Specifying the stack size of a task/thread requires trivial support (for example, in Ada it is via the `Storage_Size` attribute applied to a task; in POSIX it is via pthread attributes)
- Calculating the stack size is more difficult; as tasks enter blocks and execute procedures their stacks grow
- To estimate the maximum extent of this growth requires knowledge of the execution behaviour of each task
- This knowledge is similar to that required to undertake WCET analysis
- WCET and worst-case stack usage bounds can be obtained from control flow analysis of the task's code

Summary

- To program device drivers in high-level languages requires:
  - the ability to pass data and control information to and from the device
  - the ability to handle interrupts
- Control and data information is passed via device registers
- These are either accessed by special addresses, or via special machine instructions
- Interrupt handling requires context switching, device and interrupt identification, interrupt control, and device prioritisation
- The main requirement on a high-level language is that it provides an abstract model of device handling
- Encapsulation facilities are also required so that the non-portable code of the program can be separated from the portable part

Summary

- There are several ways to model interrupts
- In a pure shared-variable model, the driver and the device communicate using the shared device registers, and the interrupt provides condition synchronization
- Modula-1, has such a model
  - Driver processes are encapsulated in device modules which have the functionality of monitors
  - Device registers are accessed as scalar objects or arrays of bits, and an interrupt is viewed as a signal on a condition variable
- In Ada
  - Device registers can be defined as scalars and user defined record types, with a comprehensive set of facilities for mapping types onto the underlying hardware.
  - Interrupts are viewed as hardware generated procedure calls to a protected object

Summary

- Real-Time Java supports the access to memory-mapped I/O registers through the `RawMemoryClass`; however, it lacks expressive power for manipulating device registers
- Interrupts are viewed as asynchronous events.

Summary

- Low-level programming also involves the more general issue of managing the memory resources of the processor
- Ada
  - does not require a garbage collector
  - memory can be explicitly deallocated
  - the scope rules of the language allow automatic deallocation when an access types goes out of scope
  - user-defined storage pools to be defined which enable programmers to define their own memory management policies
- Real-Time Java
  - recognizes that the memory allocation policy of Java is not sustainable for real-time systems
  - allows memory to be allocated outside of the heap,
  - supports the notion of scoped memory which allows automatic reclamation of memory without garbage collection